**ASSIGNMENT 01**

**LAYERED AND HEXAGONAL ARCHITECTURES**

**SAP A.A. 2024-2025**

Casadei Giacomo – 0001005016

*Analisi del Problema*

Lo scopo di questo assignment è quello di verificare le nostre capacità nel design e implementazione di un sistema utilizzando la *layered architecture* e l’*hexagonal architecture*. In particolare, è stato richiesto di ristrutturare l’applicazione “Ebike application” demo fornita implementata come Big ball of mud in una versione basata su client-server con due backend diversi, uno costruito scegliendo una *layered architecture* e l’altro utilizzando l’*hexagonal architecture*.

*Ebike Cesena*

Indipendentemente dall’architettura scelta per il backend, sono presenti aspetti comuni tra le due soluzioni.

GUI

Immagine che contiene testo, schermata, schermo, numero

Descrizione generata automaticamente

L’applicazione presenta una parte grafica che permette all’utente di interagire col sistema. Inizialmente si presenta la schermata di Login, in cui l’utente può inserire le sue credenziali o registrarsi.

Immagine che contiene testo, schermata, software, schermo

Descrizione generata automaticamente

Successivamente la schermata principale si presenta con una sezione centrale ove è possibile visualizzare in tempo reale le bici registrate all’applicazione e le loro caratteristiche e l’utente attuale e il credito residuo.

Nella sezione superiore sono presenti vari pulsanti che consentono di utilizzare le funzioni dell’applicazione:

* Recharge credit, consente di ricaricare il proprio credito residuo inserendo la quantità da aggiungere.
* Find nearby bikes, consente di inserire le due coordinate spaziali (X e Y) e di visualizzare tutte le bici “vicine” alla posizione inserita.
* Start ride, consente di scegliere una bici e, se la bici è disponibile e il credito residuo è maggiore di 0, utilizzare la bici per effettuare una corsa fino a quando non si sceglierà di interromperla. Durante la corsa, la posizione della bici, la carica della bici e il credito residuo dell’utente verranno aggiornati in tempo reale.
* My rides, consente di visualizzare un riepilogo di tutte le corse effettuate dall’utente corrente.

Nel caso si esegua l’accesso con un utente amministratore, si renderanno visibili i pulsanti legati alla gestione del sistema All Rides, All Users e All EBikes.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Ognuno di questi pulsanti mostra una lista di tutti gli elementi corrispondenti al pulsante premuto con la possibilità di eliminarli o modificarli.

Immagine che contiene testo, elettronica, schermata, schermo

Descrizione generata automaticamente

COMUNICAZIONE CLIENT – SERVER

La comunicazione tra client e server (implementati tramite l’utilizzo di VertX) avviene tramite richieste HTTP utilizzando uno stile architetturale REST, utilizzando il formato JSON per lo scambio di informazioni strutturate.

Per propagare a tutti le modifiche effettuate/richieste da un singolo client, viene utilizzato un eventBus pubblicato su una webSocket a cui i vari client posso sottoscriversi per aggiornamenti su Utenti e Bici.

Alla ricezione di una richiesta da parte di un Client, il Server crea un Thread separato per gestire tale richiesta, mentre il Client, a seguito dell’invio di una richiesta, gestisce l’attesa di una risposta tramite l’utilizzo delle Future.

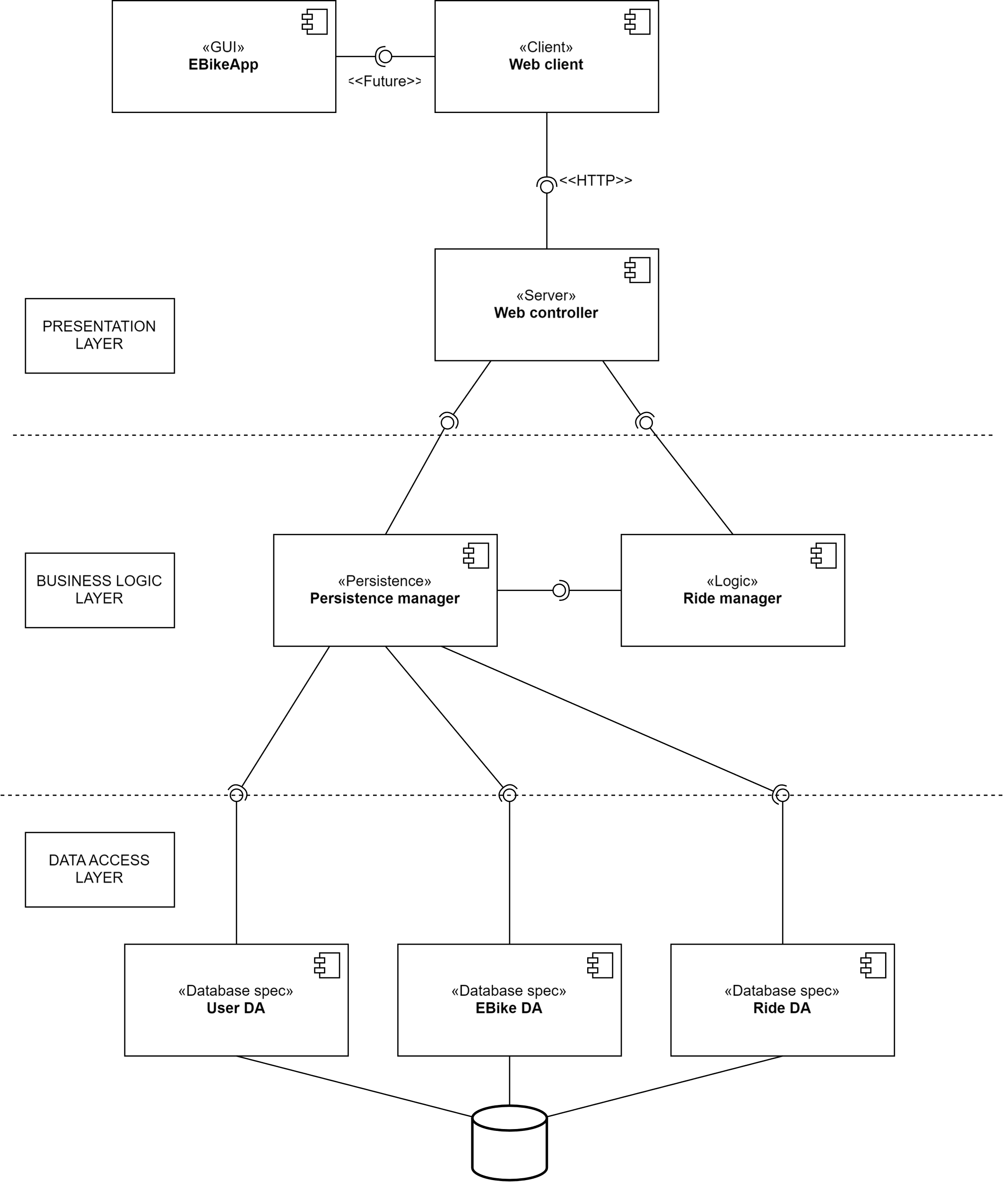
PERSISTENZA: Data Base

Entrambe le versioni del backend fanno riferimento ad un unico Data Base MySQL e le 3 tabelle di cui è composto: ebike, users, ride.

UTILS

Il progetto contiene un package chiamato “utils” che contiene classi utilizzate a tutti i livelli dell’applicazione, quali implementazioni del tipo Pair<X, Y> e Triple<X, Y, Z>, costanti per i nomi dei campi JSON utilizzati a livello web e le enumerazioni relative ai possibili stati delle bici e ai tipi di operazioni richiedibili (READ, CREATE, LOGIN …)

*LAYERED ARCHITECTURE BACKEND*

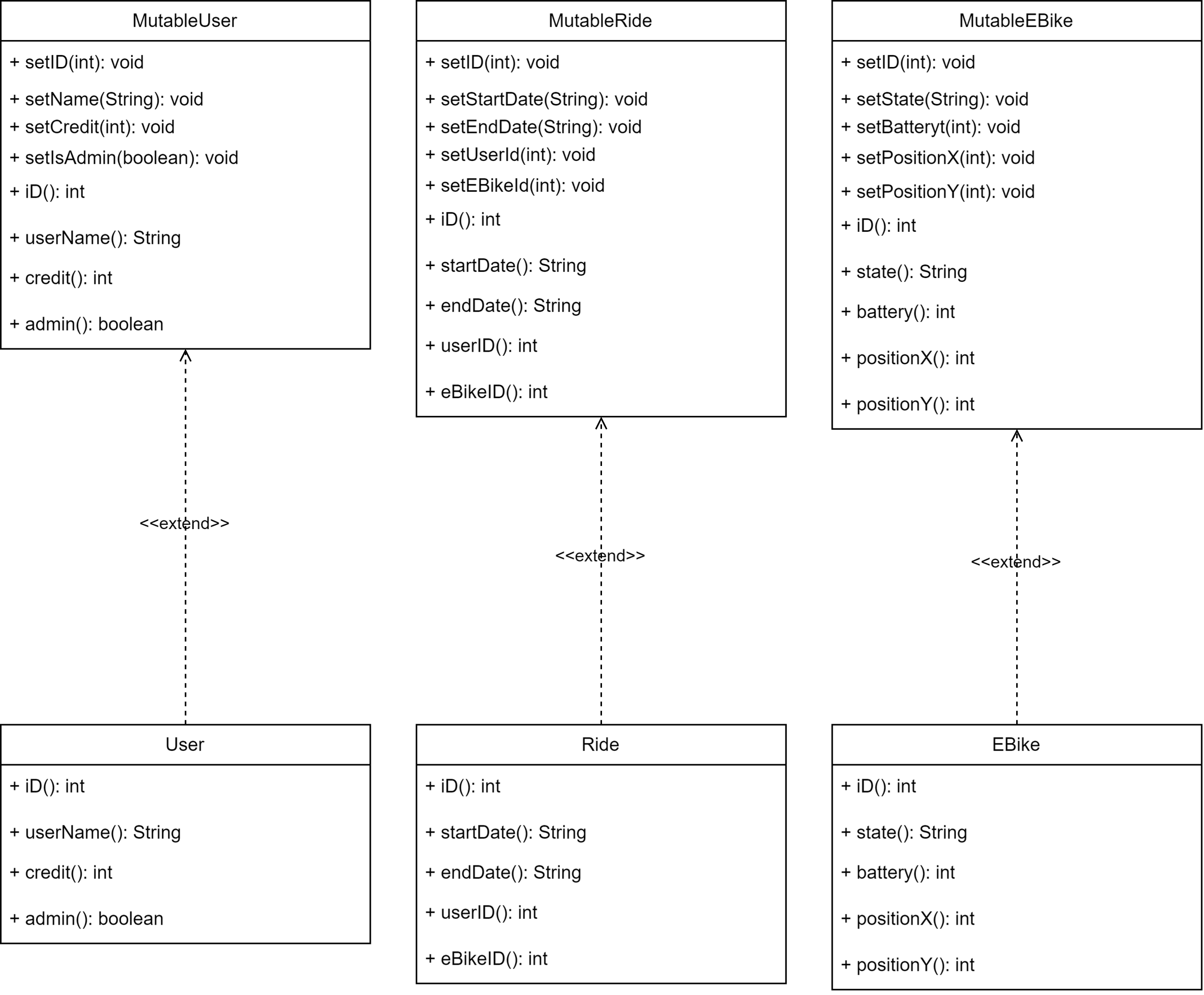
**

Ho scelto di suddividere l’applicazione in 3 layer:

* Presentation Layer: contenente la GUI ed entrambe le parti Web relative a client – server.
* Business Logic Layer: contente la logica sulla gestione delle corse (Ride manager) e il collegamento ad un eventuale sistema di persistenza (Persistence manager).
* Data Access Layer: contente le interfacce relative alla specifica implementazione del DB sottostante (MySQL)

Il Ride Manager conserva una lista delle corse in svolgimento per determinare il corretto consumo di credito, inoltre è dipendente dal Persistence Manager in quanto spesso richiede letture e update.

Entità



Ho scelto di utilizzare due implementazioni diverse per le stesse entità, quella Mutable è utilizzata nel Data Access Layer mentre la versione “Immutable” viene utilizzata nei layer rimanenti ed è un’estensione della sua controparte Mutable che espone solo i metodi di get.

Questa scelta è stata fatta per consentire al DAL più indipendenza nella mappatura dei record in oggetti utilizzabili dall’applicazione. Le funzioni di conversione sono situate nel Persistence manager.

Test

Ho testato la correttezza del Business Logic Layer e del Data Access Layer tramite una serie di test JUnit.

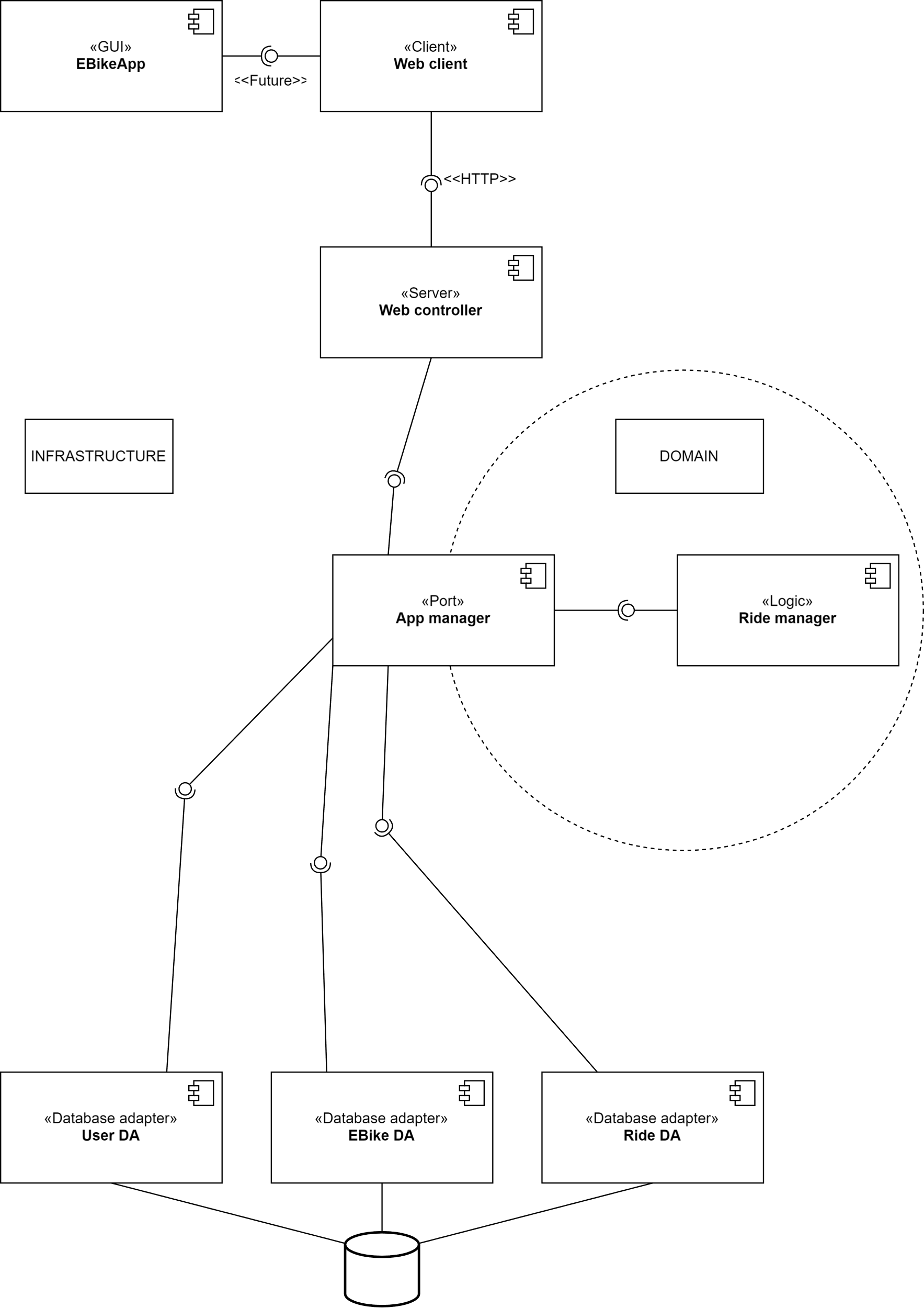
Fitness Functions

Per garantire che le dipendenze che rendono tale l’architettura Layered fosse rispettate ho scritto quattro regole utilizzando ArchUnit:

* *layeredArchitectureAccessRule*
* *presentationLayerDependsOnlyFromBusinessLayer*
* *BusinessLayerDependsOnlyFromDBLayer*
* *dataAccessLayerDoesntDependOnLayers*

Escludendo le ultime tre il cui nome è sufficientemente esplicativo riguardo a cosa vanno a testare, la prima regola definisce le classi appartenenti ai tre layer per accertarsi che nessuno possa accedere al Presentation layer, solo quest’ultimo possa accedere direttamente al Business Logic layer ed infine solo esso possa accedere al Data Access layer.

Tutte e quattro le regole restituiscono esito positivo.

*HEXAGONAL ARCHITECTURE BACKEND*

(Con lo strumento che ho utilizzato per disegnare i diagrammi gli esagoni disponibili non erano visivamente accettabili)

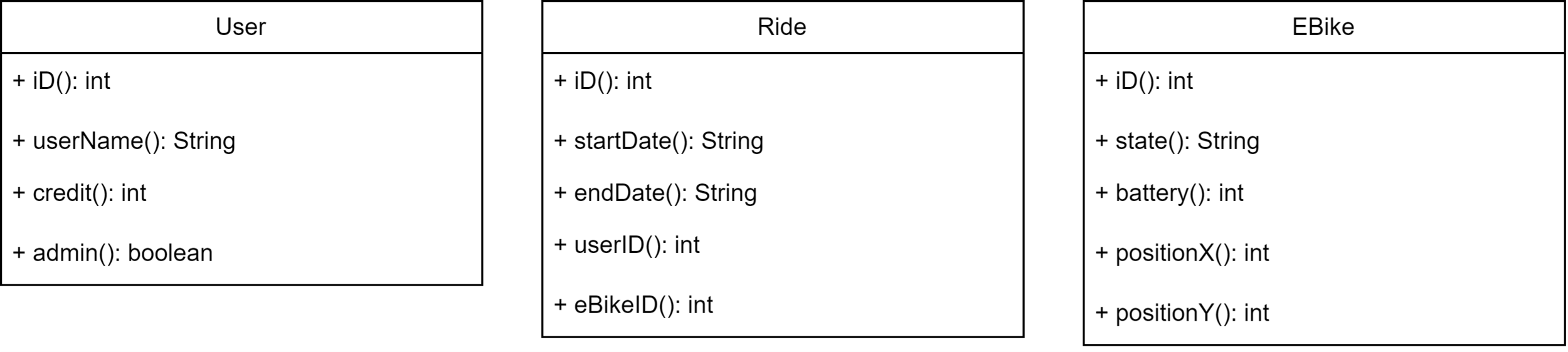
Per questa parte ho riutilizzato gran parte del codice relativo alla parte legata al DB, la GUI e la parte web client – server.

Ho scelto di suddividere l’applicazione in due esagoni:

* Domain: contiene la logica sulla gestione delle corse (Ride Manager) e la sua in/outbound port AppManager (che in parte è equivalente al Persistence manager della layered architecture)
* Infrastructure: che contiene tutto il resto, ossia gli adapter relativi al DB, la GUI e la parte Web di client – server.

A differenza dell’architettura layered, qui il Ride Manager è totalmente indipendente e stateless.

Entità



Le entità definite nel Domain sono utilizzate nel resto dell’applicazione.

Test

Avendo riutilizzato la parte di accesso ai db, ho ritenuto sufficiente testare solamente la parte di Domain, utilizzando sempre dei test JUnit.

Fitness Functions

Per verificare che i vincoli di dipendenze dell’architettura esagonale siano rispetatti, ho scritto 3 regole utilizzando ArchUnit:

* *cleanArchitectureAccessRule*
* *infrastructureLayerDependsOnlyFromDomainLayer*
* *domainLayerDoesntDependOnLayers*

La descrizione di tali regole è la medesima dell’architettura precedente.

Per rispettare il verso delle dipendenze ho utilizzato il *Dependency Injection Principle* dichiarando le interfacce relative al DB all’interno del package del Domain così che la dipendenza risulti sempre verso il centro.

Tutte e tre le regole restituiscono esito positivo.

*Quality Attributes*

Come quality attributes per entrambe le implementazioni ho scelto di concentrarmi su Performance, Scalability e Maintainability.

Performance

Per la performance ho analizzato i tempi medi di risposta per l’esecuzione di ogni possibile comando inviato dalla GUI.

QA Scenario

Context: l’applicazione consente all’utente di eseguire diverse operazioni che interagiscono con il sistema.

Stimulus: Un utente esegue una serie di operazioni, ad esempio una lettura di tutte le bici vicine alla sua posizione, una lettura di tutte le bici disponibili e l’inizio di una corsa.

Response: L’applicazione deve garantire che ogni operazione venga completata entro 1,5 secondi.

Outcome: L’utente esegue tutte le operazioni in modo fluido e senza frustrarsi per la lentezza, assicurando un’esperienza positiva.

Measure: Ho misurato i tempi medi di risposta per eseguire una di ogni operazione possibili iniziabili dall’interfaccia. Di seguito cinque medie ognuna riferita ad un’intera esecuzione di operazioni:

*Layered*: 788 - 755 - 818 - 773 – 818 per una media finale di 790 ms [Rispetta il tempo stabilito]

*Hexagonal*: 741 - 729 - 887 - 760 – 805 per una media finale di 784 ms [Rispetta il tempo stabilito]

Scalability

Per la scalability ho eseguito una serie di test per verificare le performance a fronte della lettura delle corse da parte di 100, 500 e 1000 utenti simultanei.

QA Scenario

Context: l’applicazione è progettata per gestire richieste da parte di multipli utenti che accedono contemporaneamente.

Stimulus: Un gruppo di utenti accede all'applicazione contemporaneamente per leggere informazioni.

Response: L'applicazione deve essere in grado di gestire l'aumento del numero di utenti senza degradare troppo le prestazioni. I tempi di risposta devono rimanere entro determinati limiti per determinati livelli di carico:

* **100 utenti:** Tempo di risposta massimo di 1 secondo per ogni richiesta.
* **500 utenti:** Tempo di risposta massimo di 2 secondi per ogni richiesta.
* **1000 utenti:** Tempo di risposta massimo di 4 secondi per ogni richiesta.

Outcome: L'applicazione riesce a mantenere le prestazioni desiderate all’aumentare del numero di utenti.

Measure: Di seguito tre medie ognuna riferita ad un’intera esecuzione di letture:

*Layered*:

100 utenti - 1044 - 1056 – 1059 per una media finale di 1053 ms  
500 utenti - 2201 - 2010 – 2174 per una media finale di 2128 ms  
1000 utenti - 3034 - 2898 – 2959 per una media finale di 2963 ms

*Hexagonal*:

100 utenti - 1098 - 1087 – 1128 per una media finale di 1104 ms  
500 utenti - 2397 - 2147 – 2312 per una media finale di 2385 ms  
1000 utenti - 3231 - 3140 – 3375 per una media finale di 3248 ms

A seguito delle misure, si può confermare che l’attributo risulta soddisfatto per ogni implementazione.

Maintainability

Per la maintainability mi sono limitato alla presenza di documentazione.

QA Scenario

Context: il sistema viene ripreso tra due anni e un nuovo sviluppatore vengono incaricate di implementare nuove funzionalità.

Stimulus: lo sviluppatore deve implementare le nuove funzionalità e aggiornare la documentazione.

Response: La documentazione dovrebbe essere chiara e concisa per una facile comprensione.

Outcome: Lo sviluppatore riesce in meno di un paio d’ore a individuare dove agire e aggiornare la documentazione in modo tale che rifletta le modifiche apportate.

Measure: Ho usato il plugin PMD per accertare che la percentuale di classi e metodi con la documentazione non fosse inferiore al 90%.

*Conclusione*

La maggiore difficoltà incontrata è stato il tempo concesso per l’esecuzione dell’Assignment che ho trovato un po’ ridotto soprattutto a causa del mio lavoro part-time che alterno alle lezioni. Questa limitazione mi ha costretto a tagliare aspetti che avrei voluto/dovuto includere (come l’implementazione di un Micro-Kernel per la GUI, la gestione esplicita dell’accesso concorrente al dataBase o l’utilizzo di Thread semplici invece che di strutture più raffinate come gli Executors), il che non mi soddisfa. Per la parte di “simulazione di una corsa” ho riutilizzato quasi totalmente la parte fornita dalla demo di esempio.

Escluso questo ho provato a identificare un Quality Attribute non relativo alle performance ma non sono sicuro di essere riuscito a cogliere nel segno.

Lo strumento che più ho apprezzato nell’esecuzione è stato ArchUnit, ho trovato positivamente disarmante la sua intuitività di utilizzo e la sua potenza.